

Chapter 1

SCENARIO-BASED ENGINEERING OF MULTI-AGENT SYSTEMS

Jon Whittle
QSS / NASA Ames,
jonathw@email.arc.nasa.gov

Johann Schumann
RIACS / NASA Ames,
schumann@email.arc.nasa.gov

1. Introduction

A recent development in software engineering is the paradigm of agent-oriented software engineering (AOSE) – see (Wooldridge and Ciancarini, 2001) for a survey of the state of the art. Roughly speaking, AOSE is the engineering of systems that are built from distributed, coordinating sets of agents. Most work in AOSE has been extensions of object-oriented analysis and design methodologies with emphasis placed on features that are particular to agent-based systems, such as complex coordination protocols, a high degree of concurrency and autonomy. An independent area that has also received a lot of interest is that of scenario-based software engineering (SBSE) – see (SBSE, 2000). Scenarios are traces of interactions between system components or its users. They are usually represented as abstract execution traces and serve as an interlingua between customers, system developers and test engineers. SBSE explores the ways in which scenarios can be used in the software development process and has primarily been associated with object-oriented software methodologies.

The use of scenarios is particularly suited to the development of agent-oriented systems. Such systems typically involve complex interactions between multiple coordinating agents. The interaction protocols for these systems can be very tricky to engineer. This chapter will show

how some new techniques for utilizing scenarios in software engineering, reported in (Whittle and Schumann, 2000) and extended here, can be applied to AOSE. The focus is on modeling and analyzing agent interaction protocols and excludes discussion of how to model agents' beliefs, desires and intentions. Our methodology thus aims toward applications with weak agency where agent behavior can be much more pre-determined during design time (cf. Wooldridge and Jennings, 1995; for a definition of strong agency, see ?).

Two principal ways of extending the current use of scenarios will be presented. Firstly, scenarios can be used in forward engineering, i.e., using the scenarios as a guide in developing system design models or code. Although many commercial software development environments provide forward engineering support in the form of stub code generation from design models, there is currently no available technology that can provide automated support in forward engineering design models from scenarios. In this chapter, an algorithm will be presented that can (semi-)automatically generate initial design models from a collection of scenarios. Secondly, scenarios can be used in reverse engineering, i.e., extracting a design model from existing code. The purpose of reverse engineering is to recover a faithful, high-level design of a piece of software in the case that the software source code is not available or is poorly documented. Execution scenarios can be obtained by executing an instrumented version of the code that outputs trace information to a log file. These scenarios can then be used as the basis for developing a design model.

In the context of agent-based systems, scenarios can be used to develop the following kinds of design models:

- agent interaction protocols, describing the complex interactions between agents when they are placed in the same environment;
- agent skeletons (Singh, 1998a) or abstract local descriptions of agents in terms of events that are significant for coordination with other agents;
- detailed models of the internal behavior of an agent;

In this paper, we focus on the use of statecharts Harel, 1987 to represent the design models. A statechart is a finite state machine extended by notions of hierarchy and orthogonality (for concurrent behavior). Their event-driven nature make statecharts interesting for agent-based systems. The use of scenarios in forward-engineering is already quite common in the development of object-oriented systems and forms part of many OO methodologies, namely those that are based on use cases

(Rosenberg and Scott, 1999). The idea is that use cases (which can be considered as a collection of scenarios) are used in the early stages of software design to map out the behavioral requirements of a software artifact. Scenarios, however, give a global view of a system in terms of the interactions between sub-components. For implementation, a model of the individual components is required, so the scenarios are used as a guide in developing the local model of each sub-component. Unfortunately, this transition from global scenarios to local models is currently left grossly underspecified in state of the art methodologies.

We have developed a technique for translating scenarios into behavioral models (semi-)automatically. Given a collection of scenarios, plus additional constraints that may be specified by the user, our algorithm synthesizes a behavioral model for each agent involved in the interaction. These generated models will be either interaction protocols, skeletons or detailed models, depending on the information represented in the input scenarios.

A number of other approaches have been developed for translating from scenarios to behavioral models (e.g., Khriss et al., 1999; Männistö et al., 1994; Leue et al., 1998; Somé and Dssouli, 1995), but our approach has a number of advantages, namely:

- scenarios will in general overlap. Most other approaches cannot recognize intersections between scenarios. Our approach, however, performs a *justified merging of scenarios* based on logical descriptions of the communications between agents. The communication scenarios are augmented using a constraint language and identical states in different scenarios are identified automatically based on these constraints. This leads to models both vastly reduced in size, and also corresponding more to what a human designer might produce.
- scenarios will in general conflict with each other. Our algorithm *detects and reports any conflicts* in the specification of the communications.
- the models generated by our algorithm are *highly structured*. Much of this structure is detected automatically from the communication specifications. Additional structure can be deduced from user-specified abstractions. This leads to generated models that are human-readable, reusable and maintainable, not just flat, structureless models.

The model synthesis algorithm introduced in the preceding paragraphs can also be used in a reverse engineering context. Reverse engi-

neering a model from an existing system or prototype is often a necessary process for understanding the operation of the system and how it interacts with other components. This is particularly true for agent-based systems in which agent components may exhibit emergent behaviors not explicitly called out in the software. Reverse engineering using scenarios can be undertaken as follows. Simulations of a working prototype can be instrumented to output event traces, or traces of communication events between agents. These traces, possibly abstracted to remove low-level implementation details, can be considered as scenarios. By running a series of simulations, a collection of scenarios are obtained which can then be used as input to the synthesis algorithm described in the previous paragraph. The result is a model of the interaction protocol in the existing prototype. Such a process could be used to derive abstract models of an existing system so that it could be analyzed for conformance to certain properties.

In order to present these ideas in a concrete context, the Unified Modeling Language (UML), (OMG, 2001) will be used as a language in which to express the scenarios and design models. UML is a widely used industry-standard modeling language consisting of a number of graphical notations as well as a textual constraint language, the Object Constraint Language (OCL) (Warmer and Kleppe, 1999). Recent extensions have been made to UML to better support the design of agent-based systems. UML sequence diagrams will be used to represent scenarios, statecharts will be used as design models, and class diagrams with OCL constraints will be used to give the static structure of a system. These notations will be introduced in the next section.

Section 2 introduces the basics of AOSE and SBSE. It also gives background information on UML, including an example that will be used to illustrate the algorithms in the rest of this chapter. Section 3 describes the methodology for using the automatic synthesis algorithm which is itself described in section 4. Section 5 discusses the reverse engineering aspect and conclusions are drawn in section 6.

2. Background

This section briefly describes existing work in Agent-Oriented Software Engineering (AOSE) and Scenario-based Software Engineering (SBSE). Sections 2.1 and 2.2 describe AOSE. Previous work on synthesis in SBSE is presented in section 2.3.

AOSE is concerned with the development of methodologies for engineering systems that are made up of a collection of agents. Most work in AOSE has grown out of work on object-oriented methodologies.

This is due to the fact that the development of large-scale agent-based software requires modeling methods and tools that support the entire development lifecycle. Agent-based systems are highly concurrent and distributed and hence it makes sense to employ methodologies that have already been widely accepted for distributed object-oriented systems. Indeed, agents can be viewed as “objects with attitude” (Bradshaw, 1997) and can themselves be composed out of objects. On the other hand, agents have certain features not possessed by objects — such as autonomy, the ability to act without direct external intervention; and cooperation, the ability to independently coordinate with other agents to achieve a common purpose. The precise nature of the relationship between objects and agents is as yet unclear. However, we anticipate that the use of AOSE (perhaps with further extensions) for modeling agent-based systems will increase.

For a full description of the state of the art in AOSE, see (Wooldridge and Ciancarini, 2001). The discussion here is limited to a small number of examples.

2.1 UML and AOSE

Agent UML (AUML, 2001) is an attempt to extend the Unified Modeling Language (OMG, 2001) with agent-specific concepts and notations. The premise of Agent UML is that agents are an “extension of active objects, exhibiting both dynamic autonomy (the ability to initiate action without external invocation) and deterministic autonomy (the ability to refuse or modify an external request)” (Odell et al., 2001). Agent UML attempts to augment UML to support these extensions.

In order to give an introduction to UML, an example is now presented. This example will also serve as ongoing example to illustrate the model synthesis algorithm. The example is that of an automated loading dock in which forklift agents move colored boxes between a central ramp and colored shelves such that boxes are placed on shelves of the same color. The example is presented as a case study in (Müller, 1996) of a three-layered architecture for agent-based systems, in which each agent consists of a reactive, a local planning and a coordination layer. Each layer has responsibility for certain actions: the reactive layer reacts to the environment and carries out plans sent from the planning layer; the planning layer forms plans for individual agent goals; and the coordination layer forms joint plans that require coordination between agents. We have translated part of this example into UML as a case study for our algorithm. Figure 1.1 gives the static structure of part of the system, represented as a UML class diagram. Each class can be

annotated with attributes or associations with other classes. `coordWith` describes whether an agent is currently coordinating its actions with another agent (0..1 is standard UML notation for multiplicity meaning 0 or 1), and `coordGoal` gives the current goal of this other agent. Agent interaction is based on a leader election protocol which selects an agent to delegate roles in the interaction. `leader` describes whether an agent is currently a leader. The filled diamonds in the class diagram represent aggregation (the ‘part-of’ relationship).

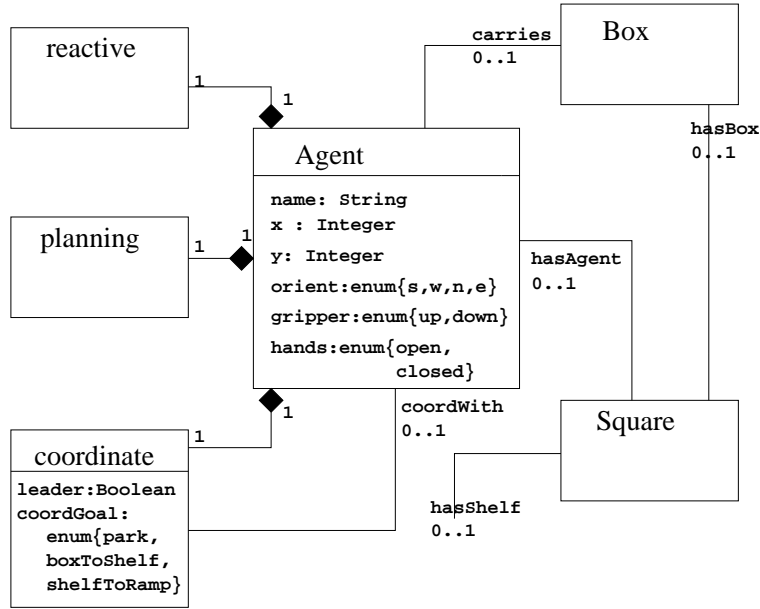


Figure 1.1. The loading dock domain

Figures 1.2, 1.3 and 1.4 are sample UML sequence diagrams (SDs) for interaction between two agents. SD1 is a failed coordination. Agent[*i*] attempts to establish a connection with Agent[*j*], but receives no response¹. So it moves around Agent[*j*]. In SD2, the move is coordinated, and SD3 shows part of a protocol for Agent[*j*] to clear a space on a shelf for Agent[*i*]. Note that these are actually *extended* sequence diagrams. ‘boxShelfToRamp’ is a sub-sequence diagram previously defined and ‘waiting’ is a state explicitly given by the user. More will be said about extended SDs in Section 4.4.2.

Agent UML (AUML) extends UML by making recommendations as to how to use UML to model agents and by offering agent-specific extensions. Work so far has concentrated on modeling agent interaction protocols. AUML suggests the use of UML sequence diagrams for mod-

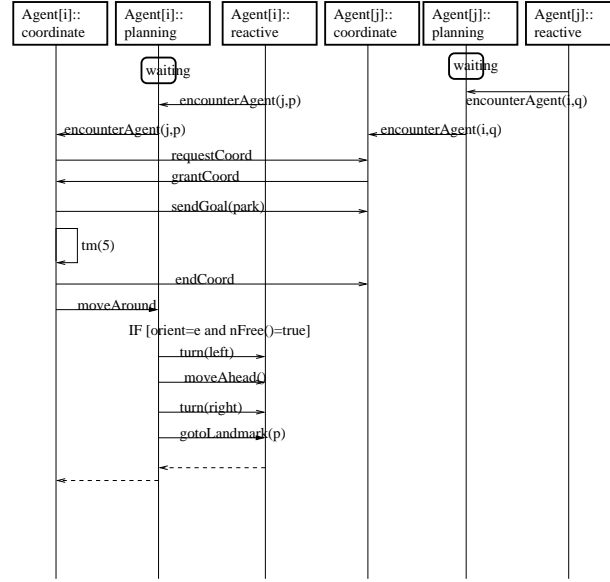


Figure 1.2. Agent Interaction (SD1).

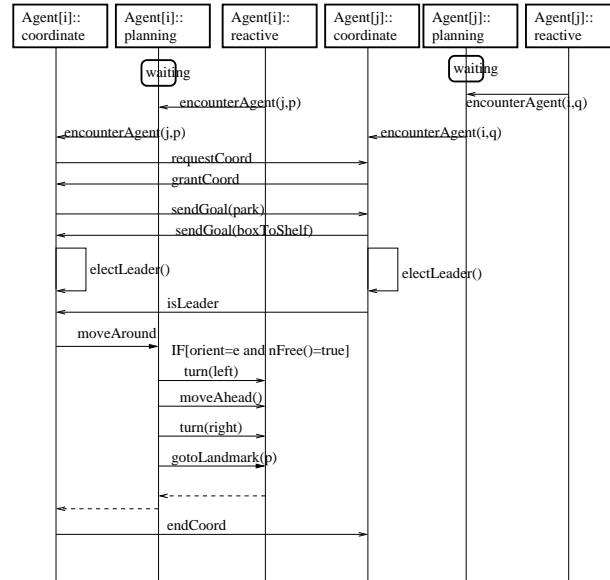


Figure 1.3. Agent Interaction (SD2).

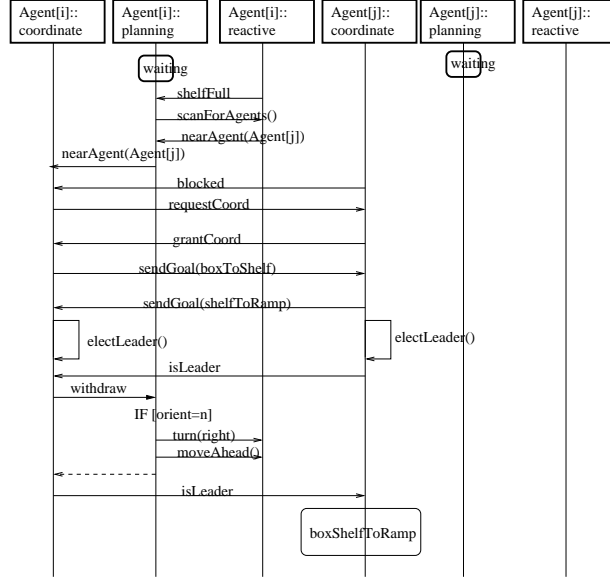


Figure 1.4. Agent Interaction (SD3).

eling interactions among agents. The main difference between AUML and UML sequence diagrams is that in UML the arrows in a sequence diagram represent messages passed between objects. In AUML, the arrows are *communication acts* between agents playing a particular role. In addition, AUML introduces additional constructs for supporting concurrent threads of execution in sequence diagrams.

Design models in UML are often expressed as UML statecharts (see Figure 1.14). A statechart is an ideal graphical notation for expressing event-driven behavior as typically found in agent-based systems. Statecharts are finite state machines (FSMs) augmented with notations for expressing hierarchy (multiple levels of states with composite states containing collections of other states) and orthogonality (composite states separated into independent modules which may run concurrently). Transitions in a statechart describe the links between different states and are labeled in the form e/a where e is an event that triggers the transition to fire and a is an action that is executed upon firing. Guards are also allowed on transitions but will not be discussed in this chapter. Statecharts are a good way of presenting large, complex finite state machines since the structuring mechanisms allow information to be hidden as necessary.

In the description of the synthesis algorithm in Section 4, interaction scenarios will be expressed as sequence diagrams, behavioral models as

statecharts — hence, the synthesis algorithm translates sequence diagrams to statecharts.

2.2 AOSE Methodologies

AUML (and similar approaches such as MESSAGE/UML (Caire et al., 2001)) describe extensions to UML and also suggest methodologies for developing agent-based systems. Other approaches also exist, however. The Gaia methodology (Wooldridge et al., 1999) advocates a process of refinement of high-level requirements into concrete implementations, and is based on the FUSION model for OO systems (Coleman et al., 1994). Gaia also includes agent-specific concepts such as *roles* that an agent may play. Each role is defined by four attributes: responsibilities, permissions, activities and protocols. Protocols are interaction patterns (with particular implementation details abstracted away) and are similar to FUSION scenarios. A protocol definition consists of a purpose, an initiator (role responsible for starting the interaction), a responder (roles with which the initiator interacts), inputs (information used by the initiator), outputs (information supplied by/to the responder during the interaction) and processing (a textual description of processing the initiator performs). Gaia has its own diagrammatic notation for expressing interaction protocols.

The final methodology we will mention is Interaction Oriented Programming (IOP) (Singh, 1998b). IOP is mainly concerned with designing and analyzing the interactions between autonomous agents. It consists of three main layers: coordination, commitments and collaboration. Coordination deals with how agents synchronize their activities. IOP specifies agents by *agent skeletons* which are abstract descriptions of agents stating only aspects that are relevant to coordination with other agents. (Singh, 1998b) describes a manual procedure for deriving agent skeletons from conversation instances between agents. Conversations are represented as Dooley graphs which are analyzed by the developer to separate out the different roles in the conversation. From this information, agent skeletons can be derived that are consistent with the conversation instances. In many ways, this approach is similar to ours of generating behavioral models from scenarios. However, our procedure is automated and has the advantages already stated in the Introduction.

2.3 From Scenarios to Behavioral Models

There have been a number of recent attempts at generating specifications from scenarios. Our work stresses the importance of obtaining a specification which can be read, understood and modified by a designer.

Many approaches make no attempt to interleave different scenarios. (van Lamsweerde, 1998) gives a learning algorithm for generating a temporal logic specification from a set of examples/counterexamples expressed as scenarios. Each scenario gives rise to a temporal logic formula G_i and scenario integration is merely $\bigcup_i G_i$ augmented with rules for identifying longest common prefixes. However, this does not correspond well to what a human designer would do, as it does not merge states lying beyond the common prefix.

A more effective integration of scenarios necessitates some way of identifying identical states in different scenarios. The solution to this in (Khriss et al., 1999) is to ask the user to explicitly name each state in the finite state machine (FSM) model generated from a scenario. Different states are then merged if they have been given the same name. This approach requires a good deal of effort from the user, however. The SCED tool (Männistö et al., 1994) generates FSMs from traces using the Biermann-Krishnaswamy algorithm (Biermann and Krishnaswamy, 1976). This algorithm uses backtracking to identify identical states in such a way that the final output FSM will be deterministic. As a result, there is no use of semantic information about the states and the algorithm ultimately may produce incorrect results by identifying two states that are in fact not the same. In addition, designers will often introduce non-determinism into their designs which will only be resolved at a later implementation stage. Hence, the insistence on determinism is overly restrictive. A successor of SCED, the MAS system (Systä, 2000), applies a highly interactive approach to the problem of identifying same states. During synthesis, MAS queries the user whether certain proposed scenarios should be integrated into the generated FSM. MAS chooses generalizations of the set of input scenarios to presented to the user in this way. In practice, however, it is likely that the user will be overwhelmed by the large number of interactive queries.

(Leue et al., 1998) tackles the problem of integration by requiring that the user gives an explicit diagram (a high-level Message Sequence Chart) showing the transitions from one scenario to the next. This merely shows, however, how the start and end points of different scenarios relate. There is no way to examine the contents of scenarios to, for example, detect interleavings or loops. (Glinz, 1995) follows a similar approach, essentially using an AND/OR tree instead of a high-level Message Sequence Chart.

The work closest to our own is described in (Somé and Dssouli, 1995) where timed automata are generated from scenarios. The user must provide message specifications with ADD and DELETE lists which maintain

a set of currently valid predicates in a STRIPS-like fashion. States are then identified if the set of valid predicates is the same.

The ability to introduce structure and hierarchy into the generated FSM is crucial if user modifications must be made. (Khriss et al., 1999) allows the limited introduction of hierarchy if the structure is explicitly represented in the scenarios (e.g., concurrent threads expressed in a collaboration diagram lead to a statechart node with two orthogonal subnodes). However, structure beyond that present in the scenarios must be introduced manually. Our work extends this approach by introducing hierarchy where the structure is deduced from other UML notations, such as a class diagram or from a domain model.

3. Forward Engineering UML Statecharts from Sequence Diagrams

An increasingly popular methodology for developing object-oriented systems is that of use case modeling (Rosenberg and Scott, 1999), in which use cases, or descriptions of the intended use of a system, are produced initially and are used as a basis for detailed design. Each use case represents a particular piece of functionality from a user perspective, and can be described by a collection of sequence diagrams. (Rosenberg and Scott, 1999) advocate developing the static model of a system (i.e., class diagram) at the same time as developing the sequence diagrams. Once this requirements phase has been completed, more detailed design can be undertaken, e.g., by producing statecharts.

This approach easily fits into popular iterative lifecycles. In contrast to the classical waterfall model where each individual design phase (requirements, design, coding, testing) is only carried out once, the phases in the iterative model are executed multiple times, until the final product is reached. Because of the focus on the final product, one can consider the software phases spiraling down to the product, hence such a process is also called a spiral model. A typical process as it might be used for an object-oriented design of agents usually has a number of phases as shown in Figure 1.5. After the inception phase where the first project ideas (and scenarios) are born, requirements are refined during the elaboration phase. In a highly iterative loop, the software is designed and implemented in this and the construction phase. Finally, efforts to finalize, test, and maintain the software are undertaken in the transition phase. Figure 1.5 also depicts which UML notations are typically used during which phase.

We leverage off this iterative approach and focus on the transition between requirements and design (elaboration and early implementation

phase). From a collection of sequence diagrams, plus information from a class diagram and user-specified constraints, a collection of statecharts is generated, one for each class (Figure 1.6). Support for iteration is extremely important — it is not expected that the designer gets the class diagram, sequence diagrams, or constraints correct first time. On the contrary, sequence diagrams will in general conflict with each other or with the constraints. The sequence diagrams can also miss important information or be ambiguous. Our methodology supports refinements, debugging, and modification of the synthesized artifacts and automatically updates the requirements accordingly. This “upwards” process (Figure 1.6) then facilitates stepwise design, refinement, and debugging of agent designs.

Although we will be focusing mainly on the synthesis part of this process, we will briefly describe how the automatic update of requirements is accomplished. Furthermore, we will demonstrate how specific parts of our approach (e.g., consistency check and introduction of hierarchy) can be used to support this iterative design process.

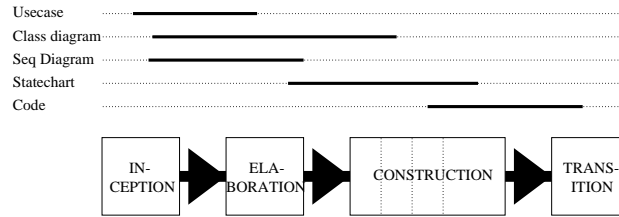


Figure 1.5. Phases of a software lifecycle and UML notations typically used during each phase

3.1 OCL specification

The lack of semantic content in sequence diagrams makes them ambiguous and difficult to interpret, either automatically or between different stakeholders. In current practice, ambiguities are often resolved by examining the informal documentation but, in some cases, ambiguities may go undetected leading to costly software errors. To alleviate this problem, we encourage the user to give pre/post-condition style OCL specifications of the messages passed between objects. OCL (Warmer and Kleppe, 1999) is part of the UML standard and is a side-effect free set-based constraint language. These OCL specifications include the declaration of *state variables*, where a state variable represents some important aspect of the system, e.g., whether or not an agent is coordi-

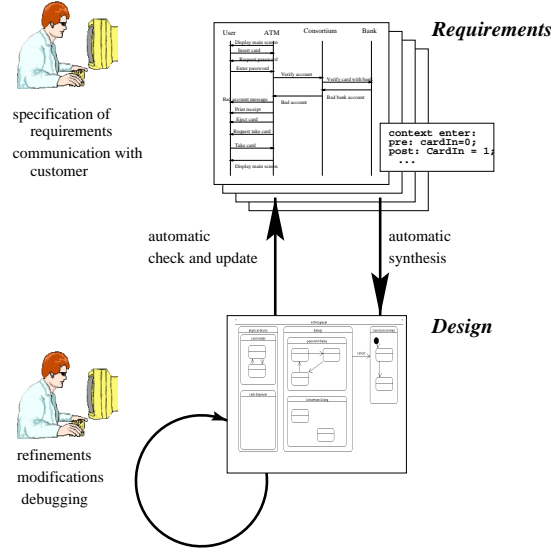


Figure 1.6. Iterative design of statecharts from requirements

nating with another agent. The OCL specifications allow the detection of conflicts between different scenarios and allow scenarios to be merged in a *justified* way. Note that not every message needs to be given a specification, although, clearly, the more semantic information that is supplied, the better the quality of the conflict detection. Currently, our algorithm only exploits constraints of the form $var = value$, but there may be something to be gained from reasoning about other constraints using an automated theorem prover or model checker.

Figure 1.7 gives specifications for selected messages in our agents example. `Agent.coordWith` has type `Agent` (it is the agent which is coordinating with `Agent`), and `Agent.coordNum`, the number of agents `Agent` is coordinating with, is a new variable introduced as syntactic sugar.

The state variables, in the form of a *state vector*, are used to characterize states in the generated statechart. The state vector is a vector of values of the state variables. In our example, the state vector for the class `coordinate` has the form:

$$\langle coordNum^{\wedge}, leader^{\wedge}, coordGoal^{\wedge} \rangle$$

where $var^{\wedge} \in Dom(var) \cup \{?\}$, and $?$ represents an unknown value. Note that since each class has a statechart, each class has its own state vector.

Our algorithm is designed to be fully automatic. The choice of the state vector, however, is a crucial design task that must be carried out

```

coordNum : enum {0,1}
leader : Boolean
coordGoal : enum {park, boxToShelf, shelfToRamp}

context Agent.coordinate::grantCoord
  pre: coordNum = 0 and coordWith.coordinate.coordNum = 0
  post: coordNum = 1 and coordWith.coordinate.coordNum = 1

context sendGoal(x : enum {park, boxToShelf, shelfToRamp})
  post: coordWith.coordinate.coordGoal = x

context electLeader
  pre: leader = false

context isLeader
  post: coordWith.coordinate.leader = true

context endCoord
  pre: coordNum = 1 and coordWith.coordinate.coordNum = 1
  post: coordNum = 0 and coordWith.coordinate.coordNum = 0
      and leader = false

```

Figure 1.7. Domain Knowledge for the Loading Dock Example

by the user. The choice of state variables will affect the generated statechart, and the user should choose state variables to reflect the parts of the system functionality that is of most interest. In this way, the choice of the state vector can be seen as a powerful abstraction mechanism — indeed, the algorithm could be used in a way that allows the user to analyze the system from a number of different perspectives, each corresponding to a particular choice of state vector.

The state variables can be chosen from information present in the class diagram. For instance, in our example, the state variables are either attributes of a particular class or based on associations. The choice is still a user activity, however, as not all attributes/associations are relevant.

4. Generating Statecharts

Synthesis of statecharts is performed in four steps: first, each SD is annotated with state vectors and conflicts with respect to the OCL specification are detected. In the second step, each annotated SD is converted into flat statecharts, one for each class in the SD. The statecharts for each class, derived from different SDs, are then merged into a single statechart for each class. Finally, hierarchy is introduced in order to enhance readability of the synthesized statecharts.

4.1 Step I: Annotating Sequence Diagrams with State Vectors

The process to convert an individual SD into a statechart starts by detecting conflicts between the SD and the OCL specification (and hence, other SDs). There are two kinds of constraints imposed on a SD: constraints on the state vector given by the OCL specification, and constraints on the ordering of messages given by the SD itself. These constraints must be solved and conflicts be reported to the user. Conflicts mean that either the scenario does not follow the user's intended semantics or the OCL specification is incorrect.

More formally, the process of conflict detection can be written as follows. An annotated sequence diagram is a sequence of messages m_1, \dots, m_n , with

$$s_1^{\text{pre}} \xrightarrow{m_1} s_1^{\text{post}}, s_2^{\text{pre}} \xrightarrow{m_2} \dots \xrightarrow{m_{r-1}} s_{r-1}^{\text{post}}, s_r^{\text{pre}} \xrightarrow{m_r} s_r^{\text{post}} \quad (1.1)$$

where the $s_i^{\text{pre}}, s_i^{\text{post}}$ are the state vectors immediately before and after message m_i is executed. S_i will be used to denote either s_i^{pre} or s_i^{post} ; $s_i^{\text{pre}}[j]$ denotes the element at position j in s_i^{pre} (similarly for s_i^{post}).

In the first step of the synthesis process, we assign values to the variables in the state vectors as shown in Figure 1.8. The variable instantiations of the initial state vectors are obtained directly from the message specifications (lines 1-5): if message m_i assigns a value y to a variable of the state vector in its pre- or post-condition, then this variable assignment is used. Otherwise, the variable in the state vector is set to an undetermined value, ?. Since each message is specified independently, the initial state vectors will contain a lot of unknown values. Most (but not all) of these can be given a value in one of two ways: two state vectors, S_i and S_j ($i \neq j$), are considered the same if they are unifiable (lines 7-8). This means that there exists a variable assignment ϕ such that $\phi(S_i) = \phi(S_j)$. This situation indicates a potential loop within a SD. The second means for assigning values to variables is the application of the frame axiom (lines 9-12), i.e., we can assign unknown variables of a pre-condition with the value from the preceding post-condition, and vice versa. This assumes that there are no hidden side-effects between messages.

A conflict (line 14) is detected and reported if the state vector immediately following a message and the state vector immediately preceding the next message differ.

Example. Figure 1.9 shows SD2 from Figure 1.3 annotated with state vectors for `Agent[i]::coordinate`. Figure 1.10 shows how the state vectors are propagated.

<p><i>Input.</i> An annotated SD</p> <p><i>Output.</i> A SD with extended annotations</p> <pre> 1 for each message m_i do 2 if m_i has a precondition $v_j = y$ 3 then $s_i^{\text{pre}}[j] := y$ else $s_i^{\text{pre}}[j] := ?$ fi 4 if m_i has a postcondition $v_j = y$ 5 then $s_i^{\text{post}}[j] := y$ else $s_i^{\text{post}}[j] := ?$ fi 6 for each state vector S do 7 if $\exists S' S' \neq S$ and some unifier ϕ with $\phi(S) = \phi(S')$ then 8 unify S_i and S_j; 9 propagate instantiations with frame axiom: 10 for each j and $i > 0$: 11 if $s_i^{\text{pre}}[j] = ?$ then $s_i^{\text{pre}}[j] := s_{i-1}^{\text{post}}[j]$ fi 12 if $s_i^{\text{post}}[j] = ?$ then $s_i^{\text{post}}[j] := s_i^{\text{pre}}[j]$ fi 13 if there is some k, l with $s_k^{\text{post}}[l] \neq s_{k+1}^{\text{pre}}[l]$ then 14 Report Conflict; 15 break;</pre>
--

Figure 1.8. Extending the state vector annotations.

4.2 Step II: Translation into a Finite State Machine

Once the variables in the state vectors have been instantiated as far as possible, a flat statechart (in fact, a finite state machine) is generated for each class (or agent) in each SD (Figure 1.11). The finite state machine for agent A is denoted by Φ_A ; its set of nodes by N_A ; its transitions by $\langle n_1, \langle \text{type}, \text{label} \rangle, n_2 \rangle$ for nodes n_1, n_2 where *type* is either *event* or *action*²; and μ_A is a function mapping a node to its state vector. \mathcal{C}_A denotes the currently processed node during the run of the algorithm. Messages directed towards a particular agent, A (i.e., $m_i^{\text{to}} = A$) are considered events in the FSM for A . Messages directed away from A (i.e., $m_i^{\text{from}} = A$) are considered actions.

The algorithm for this synthesis is depicted in Figure 1.11. Given a SD, the algorithm constructs one FSM for each agent (or for each class, in case we consider agents consisting of objects) mentioned in the sequence diagram. We start by generating a single starting node $n_{A_i}^0$ for each FSM (line 2). Then we successively add outgoing and incoming messages to the FSMs, creating new nodes as we proceed (lines 4-5).

An important step during FSM creation is the identification of loops (lines 10-13): a loop is detected if the state vector immediately after

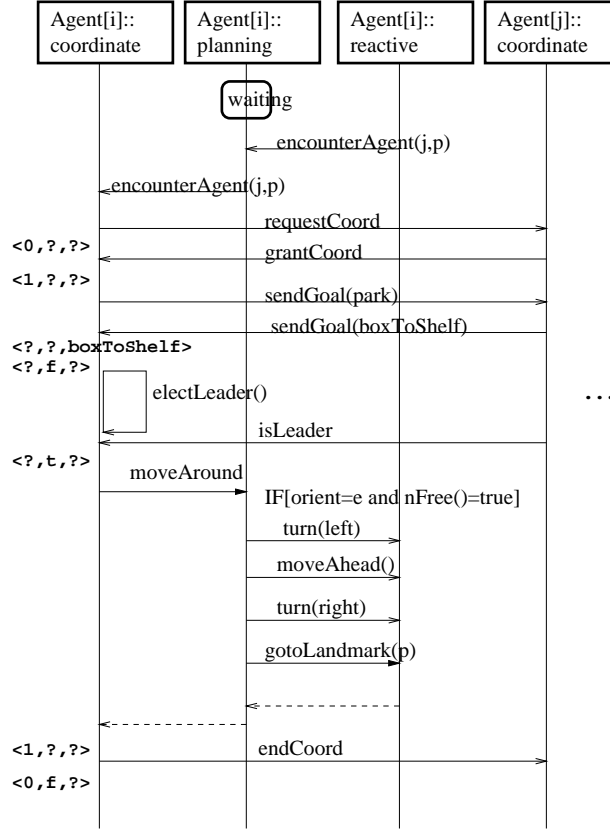


Figure 1.9. SD2 (parts) with state vectors $\langle \text{coordNum}^-, \text{leader}^-, \text{coordGoal}^- \rangle$.

the current message has been executed is the same as an existing state vector *and* if this message is state-changing, i.e., $s_i^{\text{pre}} \neq s_i^{\text{post}}$. Note that some messages may not have a specification, hence they will not affect the state vector. To identify states based solely on the state vector would result in incorrect loop detection.

4.3 Step III: Merging multiple Sequence Diagrams

The previous steps concerned the translation of a single SD into a number of statecharts, one for each class. Once this is done for each SD, there exists a collection of flat statecharts for each class. We now show how the statecharts for a particular class can be merged.

Merging statecharts derived from different SDs is based upon identifying *similar* states in the statecharts. Two nodes of a statechart are

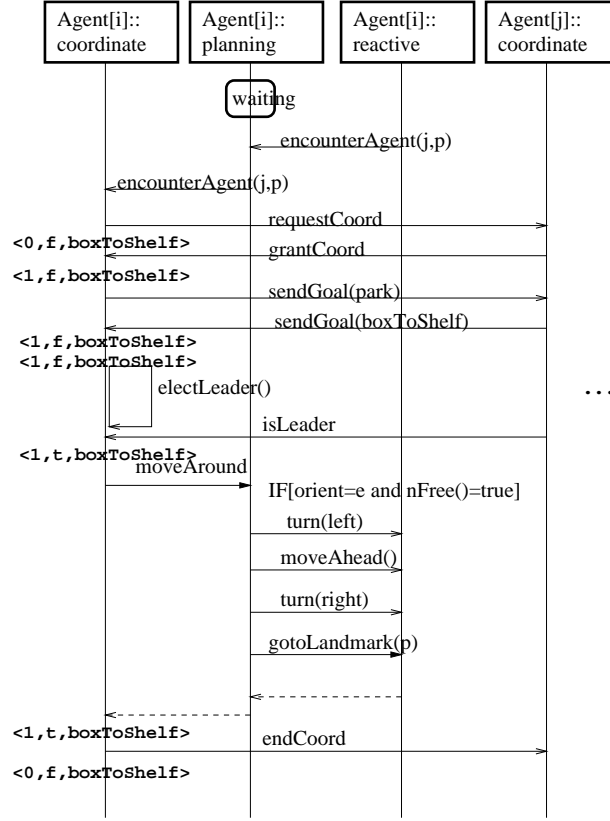


Figure 1.10. SD2 after extension of state vector annotations with state vectors for $\text{Agent}[i]::\text{coordinate}$

similar if they have the same state vector and they have at least one incoming transition with the same label. The first condition alone would produce an excessive number of similar nodes since some messages do not change the state vector. The existence of a common incoming transition which we require in addition means that in both cases, an event has occurred which leaves the state variables in an identical assignment. Hence, our definition of similarity takes into account the ordering of the messages and the current state. Figure 1.12 shows how two nodes with identical state vector S and incoming transitions labeled with l can be merged together.

The process of merging multiple statecharts proceeds as follows: we generate a new statechart and connect its initial node by empty ϵ -transitions with the initial nodes of the individual statecharts derived from each SD. Furthermore, all pairs of nodes which are *similar* to each

Input. A SD, S , with agents A_1, \dots, A_k and messages m_1, \dots, m_r
Output. A FSM Φ_{A_i} for each agent, $1 \leq i \leq k$.

```

1 for  $i = 1, \dots, k$  do
2   Create a FSM,  $\Phi_{A_i}$ , with an initial node,  $n_{A_i}^0$ ;  $\mathcal{C}_{A_i} := n_{A_i}^0$ ;
3 for  $i = 1, \dots, r$  do
4    $\text{ADD}(m_i, \text{action}, m_i^{\text{from}})$ ;
5    $\text{ADD}(m_i, \text{event}, m_i^{\text{to}})$ ;
6 where  $\text{ADD}(\text{mess } m_i, \text{type } t, \text{agent } A)$ 
7   if there is a node  $n \in N_A$ , a transition  $\langle \mathcal{C}_A, \langle t, m_i \rangle, n \rangle$ 
8     and  $s_i^{\text{post}} = \mu_A(n)$  then
9      $\mathcal{C}_A := n$ ;
10  else if there is  $n \in N_A$  with  $s_i^{\text{post}} = \mu_A(n)$ 
11    and  $m_i$  is state-changing then
12    add new transition  $\langle \mathcal{C}_A, \langle t, m_i \rangle, n \rangle$ ;
13     $\mathcal{C}_A := n$ ;
14  else
15    add a new node  $n$  and let  $\mu_A(n) := s_i^{\text{post}}$ ;
16    add transition  $\langle \mathcal{C}_A, \langle t, m_i \rangle, n \rangle$ ;
17     $\mathcal{C}_A := n$ ;

```

Figure 1.11. Translating a sequence diagram into FSMs.

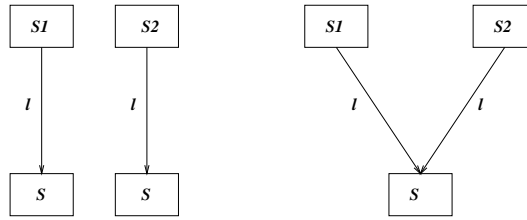


Figure 1.12. Merging of similar states (before and after the merge).

other are connected by ϵ -transitions. Then we remove ϵ -transitions, and resolve many non-deterministic branches. For this purpose, we use an algorithm which is a variant of the standard algorithm for transforming a non-deterministic finite automaton into a deterministic finite automaton (Aho et al., 1986). The output of the algorithm is only deterministic in that there are no ϵ -transitions remaining. There may still be two transitions leaving a state labelled with the same events but different actions. Hence, our algorithm may produce non-deterministic statecharts, which allows a designer to refine the design later.

4.4 Step IV: Introducing Hierarchy

So far, we have discussed the generation of flat finite state machines. In practice, however, statechart designs tend to get very large. Thus, the judicious use of hierarchy is crucial to the readability and maintainability of the designs. Highly structured statecharts do not only facilitate clean presentation of complex behavior on the small computer screen, but also emphasize major design decisions. Thus, a clearly structured statechart is easier to understand and refine in an iterative design process. Our approach provides several ways for introducing hierarchy into the generated FSMs. In this chapter, we will discuss methods which use information contained in the state vectors, in associated UML class diagrams, or explicitly given by the user (in the form of preferences and extended sequence diagrams).

4.4.1 The State Vector as a Basis for Hierarchy.

State variables usually encode that the system is in a specific mode or state (e.g., agent is the leader or not). Thus, it is natural to partition the statechart into subcharts containing all nodes belonging to a specific mode of the system. More specifically, we recursively partition the set of nodes according to the different values of the variables in the state vectors. In general, however, there are many different ways of partitioning a statechart, not all of them suited for good readability. We therefore introduce additional heuristic constraints (controlled by the user) on the layout of the statechart:

The maximum depth of hierarchy (d_{max}): Too many nested levels of compound states limit readability of the generated statechart. On the other hand, a statechart which is too flat will contain very large compound nodes, making reading and maintaining the statechart virtually impossible.

The maximum number of states on a single level ($N_{max}(d)$): This constraint is somewhat orthogonal to the first one and also aims at generating “handy” statecharts.

The maximum percentage of inter-level transitions: Transitions between different levels of the hierarchy usually limit modularity, but occasionally they can be useful. Thus their relative number should be limited (usually to around 5–10%).

A partial ordering over the state variables (\prec): This ordering describes the sequence in which partitions should be attempted. It

provides a means to indicate that some state variables may be more “important” than others and thus should be given priority. This ordering encapsulates important design decisions about how the statechart should be split up.

In general, not all of the above constraints can be fulfilled at the same time. Therefore our algorithm has the capability to do a search for an optimal solution. This search is done using backtracking over different variable sequences ordered with respect to \prec .

The process of structuring a given subset S of the nodes of a generated FSM is shown in Figure 1.13. Given a subset of variables W of the state vector over which to partition and a (partial) ordering \prec , a sequence W' is constructed with respect to the ordering \prec . Then the nodes S are partitioned recursively according to the variable sequence W' . Let v_j be the top-level variable (minimal in W') on which to split (line 11). The partition is made up of m equivalence classes corresponding to each possible value of v_j given in the SDs. Before we actually perform the split, we check if the constraints hold (lines 10 and 16). Only then is the set of nodes split and the algorithm descends recursively (line 17). After all node sets have been partitioned, we levelwise assemble all non-empty partitions (line 19). Once this algorithm terminates, we check if it is a “good” hierarchy with respect to our criteria (line 21). Because some of the constraints (e.g., number of interlevel transitions) can only be checked globally, we have to perform these tests after the partitioning.

In case the partition does not meet our design criteria described, a warning will be issued that the given ordering would result in a non-optimal hierarchy and a new ordering of the variables is selected. This selection is done until the criteria are met.

Example. Figure 1.14 gives a partitioned statechart for agent communication generated from SD1, SD2 and SD3. The flat statechart was first split over `coordNum`, followed by `leader` and finally `coordGoal` (i.e. `coordNum` \prec `leader` \prec `coordGoal`).

4.4.2 Extended Sequence Diagrams.

Other authors (Gehrke and Firley, 1999; Breu et al., 1998) have already noted that the utility of sequence diagrams to describe system behavior could be vastly increased by extending the notation. A basic SD supports the description of *exemplary* behavior — one concrete interaction — but when used in requirements engineering, a *generative* style is more appropriate, in which each SD represents a collection of interactions. Extensions that have been suggested include the ability to allow **case** statements, loops and sub-SDs. We go further than this

Input. A FSM with nodes N , state vector mapping, μ , ordering \prec over a subset $W \subset V$ of the state variables, and subset of N , $S \subset N$.
Output. A partitioning \mathcal{P} of the FSM

```

1  $W' := \langle v_1, \dots, v_k \rangle$  for  $v_i \in W$  and  $v_i \prec v_j, i < j$ ;  $ok := \text{TRUE}$ ;
2 do
3    $\mathcal{P} := \text{PARTITION}(S, W', 1)$ ;           // partition this set
4 while  $\neg ok \wedge \neg \text{OPTIMAL}(\mathcal{P})$  do
5    $ok := \text{TRUE}$ ;
6    $W' := \text{select-new-variable-ordering}(W)$ ;
7    $\mathcal{P} := \text{PARTITION}(S, W')$ ;
8 done
9 where  $\text{PARTITION}(S, W', d)$ 
10  if  $(d > d_{max} \wedge |S| < N_{min}(d))$   $ok := \text{FALSE}$ ;
11   $v_j := \text{first}(W')$ ;           // split on first var. in  $W'$ 
12   $\mathcal{D}_S(v_j) := \bigcup_{s \in S} \{\mu(s)[j]\}$ ;
13   $m := |\mathcal{D}_S(v_j)|$ ;           //  $m$  is number of partitions
14  for  $1 \leq i \leq m$  do           // on the current level
15     $S_i := \{s \in S \mid \mu(s)[j] = \text{ith}(\mathcal{D}_S(v_j))\}$ ;
16    if  $(|S_i| > N_{max}(d))$   $ok := \text{FALSE}$ ;
17     $\mathcal{P}_i := \text{PARTITION}(S_i, \text{rest}(W'))$ ; // call the partitioning
18  done           // recursively
19   $\mathcal{P} := \langle \mathcal{P}_i \mid \mathcal{P}_i \neq \langle \rangle \rangle$  // assemble result
20 where  $\text{OPTIMAL}(\mathcal{P})$ 
21  check  $\mathcal{P}$  according to our design criteria

```

Figure 1.13. Sketch of algorithm for partitioning over the state vector

and advocate the use of language constructs that allow behavior to be generalized. Example constructs we have devised so far include:

- *any_order*(m_1, \dots, m_n): specify that a group of messages may occur in any order;
- *or*(m_1, \dots, m_n): a message may be any one of a group of messages;
- *generalize*(m, SubSD): a message gives the same behavior when sent/received at any point in the sub-sequence diagram;
- *allInstances*(m, I): send a message to all instances in I ;

These constructs also suggest ways of introducing structure into the generated statecharts. As an example, *any_order*(m_1, \dots, m_n) can be implemented by n concurrent statecharts (see Figure 1.15), connected by the UML synchronization operator (the black bar) which waits until

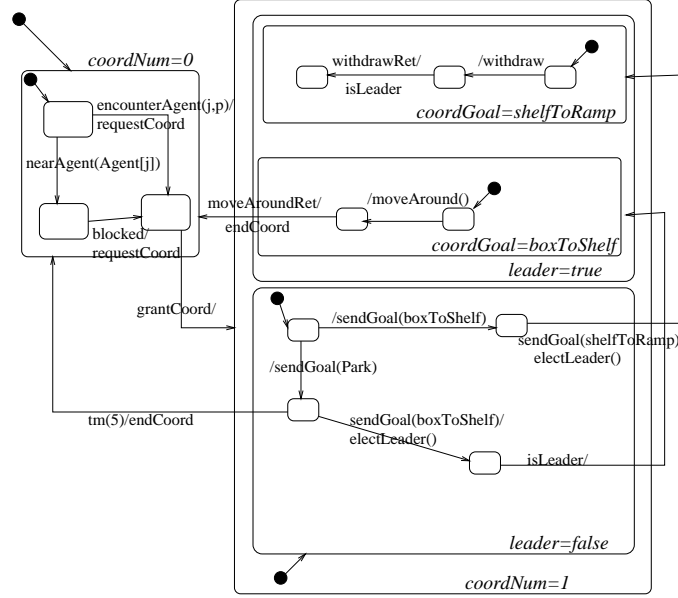


Figure 1.14. Hierarchical Statechart for Agent::coordinate.

all its source states are entered before its transition is taken. This is particularly useful if m_1, \dots, m_n are not individual messages, but sub-sequence diagrams. Figure 1.15 also shows how the other constructs mentioned above can be implemented as statecharts. *allInstances* is implemented by a local variable that iterates through each instance, and sends the message to that instance.

Example. These extensions of the SDs are convenient if, for example, our agent design requires an emergency shutdown. When activated, it sends the message *emergency* to each agent. This can be expressed as *allInstances(emergency, Agent::coordinate)*. If an agent is coordinating with other agents (regardless if it is the leader or not), a fail-safe state needs to be entered. Such a behavior is shown in Figure 1.16 and has been expressed conveniently as *generalize(emergency, (SD-describing-the-coordinated-behavior))*. This specification makes a number of sequence diagrams superfluous in which the emergency message is received in different situations.

Similarly, in situations where several parts of an interagent communication require no specific order, a compact way of specification can be used, for example, *any_order(inquire_box_color, inquire_box_size)*.

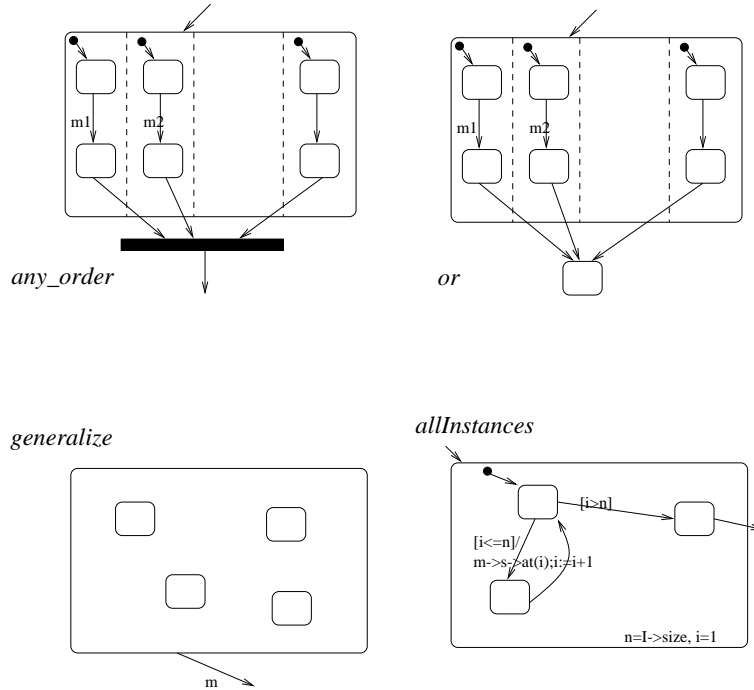


Figure 1.15. Hierarchy by Macro Commands

4.4.3 Class Diagrams.

During the synthesis process, it is also important to incorporate other design decisions made by the developer. Within the UML framework, a natural place for high-level design decisions is the *class diagram*. It describes the types of the objects in the system and the static relationships among them.

A hierarchical structure of a generated statechart can easily be obtained from the class diagram: the outermost superstate (surmounting the entire statechart) corresponds to the class node of the corresponding object. Aggregation results in a grouping of nodes. If a class contains several sub-classes, the statecharts corresponding to the sub-classes are sub-nodes of the current node.

This way of introducing structure is somewhat higher-level than the first two. Typically, the class diagram can be used to obtain a very abstract structure and the method described above (using state variables) can be used to introduce further structure within each subchart.

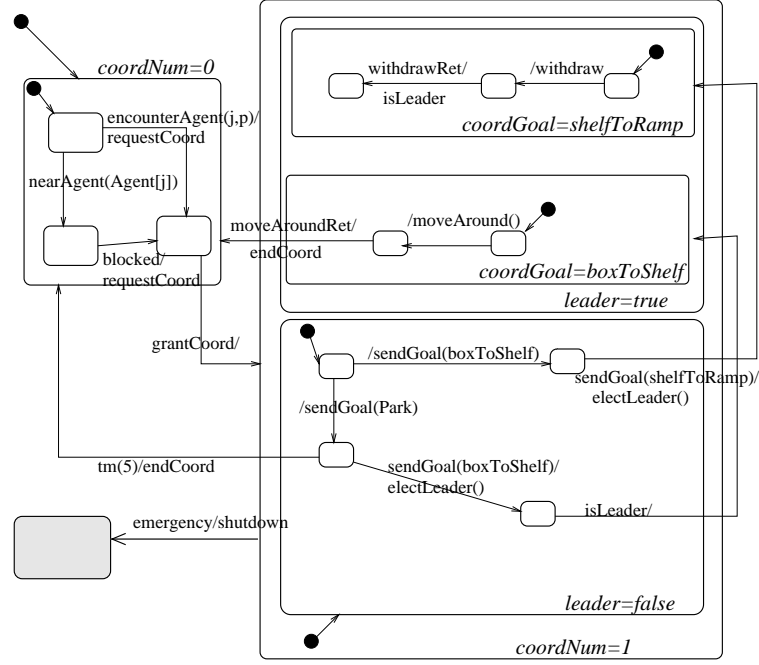


Figure 1.16. Hierarchical Statechart for Agent::coordinate with emergency handling (extension of Figure 1.14)

4.5 Statechart Layout and Hierarchy

For practical usability and readability of the synthesized statechart, a good layout is extremely important. Only then can automatic generation of agent designs and skeletons be accepted by the practitioner. Our current prototype system does not provide sophisticated layout generation³. There is a substantial body of work on automatic layout of graphs (e.g., (Battista et al., 1999)). In particular, (Castellø et al., 2000a) has developed algorithms for automatic positioning of the elements of a hierarchical statechart on the canvas.

Generation of a graph layout is subject to a number of constraints. The most important constraints concern spacing of the graph nodes (how much space does each element require?), routing of the transitions and labeling. A good layout prescribes that the arrows, representing transitions don't cross nodes and other transitions too often. On the other hand, transitions which are too long (e.g., moving around several nodes) reduce readability. Furthermore, the layout algorithm has to take into account that labels on the transitions must be placed carefully such

that they do not overlap. For our tool, we are investigating algorithms like (Castellø et al., 2000a; Castellø et al., 2000b), techniques coming from VLSI design (Bhatt and Leighton, 1984) and layout/labeling of topographical maps.

4.6 Multi-View Design

The automatic introduction of hierarchy provides another benefit: multiple views on the design. Since all important information is already contained in the flat statechart, changing hierarchy and layout does not affect the behavior of the system. Therefore, the user can, as described above, set individual preferences on how to structure and display the design. This feature opens up the possibility to keep multiple different hierarchies of the same design within the system at the same time. So, for example, different software engineers, working on different aspects of the agent could work with different hierarchies. Each designer would select a hierarchy which displays his/her focus of interest in a compact way (e.g., on top of the hierarchy in a single supernode). Also, during different stages of the software cycle, different hierarchies can be suitable, e.g., for design, for generation of test cases (e.g., involving the specific values of the state variables), or for debugging (Schumann, 2000). This multi-hierarchy approach exhibits some similarities to defining views in a relational database or individual formatting options within a word-processor.

5. Discussion: Reverse Engineering of Agent Communication Protocols

Section 4 described the use of scenarios in forward engineering agent-based systems. As described in the Introduction, however, scenarios can also be utilized in reverse engineering. In practical software engineering, legacy code often exists which has to be integrated into a new system. However, in many cases, the code is poorly documented. Before modification or integration of the code can be attempted, reverse engineering needs to be performed, in order to understand how that piece of code works — a time-consuming task. In particular, for agent-based systems, understanding the operation of distributed pieces of software is particularly troublesome.

In the framework of transforming requirements into designs presented in this chapter, reverse engineering is also supported. Given a software artifact that can be executed, the code is run on a number of relevant test cases. In general, such test cases give only the input/output behavior of the code and not the internal execution steps. By instrumenting the

```

graph TD
    TC[TESTCASES input/output] --> SA((Software artifact  
instrumented to output communications))
    SA --> ET[Event traces  
sequence diagrams]
    ET --> SA
    ET --> SA2[ ]
    SA2 --> SA3[ ]
    SA3 --> SA4[ ]
    SA4 --> SA5[ ]
    SA5 --> SA6[ ]
    SA6 --> SA7[ ]
    SA7 --> SA8[ ]
    SA8 --> SA9[ ]
    SA9 --> SA10[ ]
    SA10 --> SA11[ ]
    SA11 --> SA12[ ]
    SA12 --> SA13[ ]
    SA13 --> SA14[ ]
    SA14 --> SA15[ ]
    SA15 --> SA16[ ]
    SA16 --> SA17[ ]
    SA17 --> SA18[ ]
    SA18 --> SA19[ ]
    SA19 --> SA20[ ]
    SA20 --> SA21[ ]
    SA21 --> SA22[ ]
    SA22 --> SA23[ ]
    SA23 --> SA24[ ]
    SA24 --> SA25[ ]
    SA25 --> SA26[ ]
    SA26 --> SA27[ ]
    SA27 --> SA28[ ]
    SA28 --> SA29[ ]
    SA29 --> SA30[ ]
    SA30 --> SA31[ ]
    SA31 --> SA32[ ]
    SA32 --> SA33[ ]
    SA33 --> SA34[ ]
    SA34 --> SA35[ ]
    SA35 --> SA36[ ]
    SA36 --> SA37[ ]
    SA37 --> SA38[ ]
    SA38 --> SA39[ ]
    SA39 --> SA40[ ]
    SA40 --> SA41[ ]
    SA41 --> SA42[ ]
    SA42 --> SA43[ ]
    SA43 --> SA44[ ]
    SA44 --> SA45[ ]
    SA45 --> SA46[ ]
    SA46 --> SA47[ ]
    SA47 --> SA48[ ]
    SA48 --> SA49[ ]
    SA49 --> SA50[ ]
    SA50 --> SA51[ ]
    SA51 --> SA52[ ]
    SA52 --> SA53[ ]
    SA53 --> SA54[ ]
    SA54 --> SA55[ ]
    SA55 --> SA56[ ]
    SA56 --> SA57[ ]
    SA57 --> SA58[ ]
    SA58 --> SA59[ ]
    SA59 --> SA60[ ]
    SA60 --> SA61[ ]
    SA61 --> SA62[ ]
    SA62 --> SA63[ ]
    SA63 --> SA64[ ]
    SA64 --> SA65[ ]
    SA65 --> SA66[ ]
    SA66 --> SA67[ ]
    SA67 --> SA68[ ]
    SA68 --> SA69[ ]
    SA69 --> SA70[ ]
    SA70 --> SA71[ ]
    SA71 --> SA72[ ]
    SA72 --> SA73[ ]
    SA73 --> SA74[ ]
    SA74 --> SA75[ ]
    SA75 --> SA76[ ]
    SA76 --> SA77[ ]
    SA77 --> SA78[ ]
    SA78 --> SA79[ ]
    SA79 --> SA80[ ]
    SA80 --> SA81[ ]
    SA81 --> SA82[ ]
    SA82 --> SA83[ ]
    SA83 --> SA84[ ]
    SA84 --> SA85[ ]
    SA85 --> SA86[ ]
    SA86 --> SA87[ ]
    SA87 --> SA88[ ]
    SA88 --> SA89[ ]
    SA89 --> SA90[ ]
    SA90 --> SA91[ ]
    SA91 --> SA92[ ]
    SA92 --> SA93[ ]
    SA93 --> SA94[ ]
    SA94 --> SA95[ ]
    SA95 --> SA96[ ]
    SA96 --> SA97[ ]
    SA97 --> SA98[ ]
    SA98 --> SA99[ ]
    SA99 --> SA100[ ]
    SA100 --> SA101[ ]
    SA101 --> SA102[ ]
    SA102 --> SA103[ ]
    SA103 --> SA104[ ]
    SA104 --> SA105[ ]
    SA105 --> SA106[ ]
    SA106 --> SA107[ ]
    SA107 --> SA108[ ]
    SA108 --> SA109[ ]
    SA109 --> SA110[ ]
    SA110 --> SA111[ ]
    SA111 --> SA112[ ]
    SA112 --> SA113[ ]
    SA113 --> SA114[ ]
    SA114 --> SA115[ ]
    SA115 --> SA116[ ]
    SA116 --> SA117[ ]
    SA117 --> SA118[ ]
    SA118 --> SA119[ ]
    SA119 --> SA120[ ]
    SA120 --> SA121[ ]
    SA121 --> SA122[ ]
    SA122 --> SA123[ ]
    SA123 --> SA124[ ]
    SA124 --> SA125[ ]
    SA125 --> SA126[ ]
    SA126 --> SA127[ ]
    SA127 --> SA128[ ]
    SA128 --> SA129[ ]
    SA129 --> SA130[ ]
    SA130 --> SA131[ ]
    SA131 --> SA132[ ]
    SA132 --> SA133[ ]
    SA133 --> SA134[ ]
    SA134 --> SA135[ ]
    SA135 --> SA136[ ]
    SA136 --> SA137[ ]
    SA137 --> SA138[ ]
    SA138 --> SA139[ ]
    SA139 --> SA140[ ]
    SA140 --> SA141[ ]
    SA141 --> SA142[ ]
    SA142 --> SA143[ ]
    SA143 --> SA144[ ]
    SA144 --> SA145[ ]
    SA145 --> SA146[ ]
    SA146 --> SA147[ ]
    SA147 --> SA148[ ]
    SA148 --> SA149[ ]
    SA149 --> SA150[ ]
    SA150 --> SA151[ ]
    SA151 --> SA152[ ]
    SA152 --> SA153[ ]
    SA153 --> SA154[ ]
    SA154 --> SA155[ ]
    SA155 --> SA156[ ]
    SA156 --> SA157[ ]
    SA157 --> SA158[ ]
    SA158 --> SA159[ ]
    SA159 --> SA160[ ]
    SA160 --> SA161[ ]
    SA161 --> SA162[ ]
    SA162 --> SA163[ ]
    SA163 --> SA164[ ]
    SA164 --> SA165[ ]
    SA165 --> SA166[ ]
    SA166 --> SA167[ ]
    SA167 --> SA168[ ]
    SA168 --> SA169[ ]
    SA169 --> SA170[ ]
    SA170 --> SA171[ ]
    SA171 --> SA172[ ]
    SA172 --> SA173[ ]
    SA173 --> SA174[ ]
    SA174 --> SA175[ ]
    SA175 --> SA176[ ]
    SA176 --> SA177[ ]
    SA177 --> SA178[ ]
    SA178 --> SA179[ ]
    SA179 --> SA180[ ]
    SA180 --> SA181[ ]
    SA181 --> SA182[ ]
    SA182 --> SA183[ ]
    SA183 --> SA184[ ]
    SA184 --> SA185[ ]
    SA185 --> SA186[ ]
    SA186 --> SA187[ ]
    SA187 --> SA188[ ]
    SA188 --> SA189[ ]
    SA189 --> SA190[ ]
    SA190 --> SA191[ ]
    SA191 --> SA192[ ]
    SA192 --> SA193[ ]
    SA193 --> SA194[ ]
    SA194 --> SA195[ ]
    SA195 --> SA196[ ]
    SA196 --> SA197[ ]
    SA197 --> SA198[ ]
    SA198 --> SA199[ ]
    SA199 --> SA200[ ]
    SA200 --> SA201[ ]
    SA201 --> SA202[ ]
    SA202 --> SA203[ ]
    SA203 --> SA204[ ]
    SA204 --> SA205[ ]
    SA205 --> SA206[ ]
    SA206 --> SA207[ ]
    SA207 --> SA208[ ]
    SA208 --> SA209[ ]
    SA209 --> SA210[ ]
    SA210 --> SA211[ ]
    SA211 --> SA212[ ]
    SA212 --> SA213[ ]
    SA213 --> SA214[ ]
    SA214 --> SA215[ ]
    SA215 --> SA216[ ]
    SA216 --> SA217[ ]
    SA217 --> SA218[ ]
    SA218 --> SA219[ ]
    SA219 --> SA220[ ]
    SA220 --> SA221[ ]
    SA221 --> SA222[ ]
    SA222 --> SA223[ ]
    SA223 --> SA224[ ]
    SA224 --> SA225[ ]
    SA225 --> SA226[ ]
    SA226 --> SA227[ ]
    SA227 --> SA228[ ]
    SA228 --> SA229[ ]
    SA229 --> SA230[ ]
    SA230 --> SA231[ ]
    SA231 --> SA232[ ]
    SA232 --> SA233[ ]
    SA233 --> SA234[ ]
    SA234 --> SA235[ ]
    SA235 --> SA236[ ]
    SA236 --> SA237[ ]
    SA237 --> SA238[ ]
    SA238 --> SA239[ ]
    SA239 --> SA240[ ]
    SA240 --> SA241[ ]
    SA241 --> SA242[ ]
    SA242 --> SA243[ ]
    SA243 --> SA244[ ]
    SA244 --> SA245[ ]
    SA245 --> SA246[ ]
    SA246 --> SA247[ ]
    SA247 --> SA248[ ]
    SA248 --> SA249[ ]
    SA249 --> SA250[ ]
    SA250 --> SA251[ ]
    SA251 --> SA252[ ]
    SA252 --> SA253[ ]
    SA253 --> SA254[ ]
    SA254 --> SA255[ ]
    SA255 --> SA256[ ]
    SA256 --> SA257[ ]
    SA257 --> SA258[ ]
    SA258 --> SA259[ ]
    SA259 --> SA260[ ]
    SA260 --> SA261[ ]
    SA261 --> SA262[
```

Figure 1.17. Extracting Behavioral Models of Agent Systems using Scenarios

For this approach to work in practice, a number of important issues need to be addressed. In what follows, \mathcal{P} will abbreviate the existing software artifact and \mathcal{T} will stand for the set of execution traces produced by the instrumented code.

- \mathcal{P} must be instrumented so that the appropriate scenarios can be collected from the executions of each test case. In initial experiments, existing Java code is being reverse engineered. Whilst, in general, code that needs to be reverse engineered is probably legacy code from decades ago (e.g., written in COBOL), it is likely

that code in agent-based systems will often be written in Java. The need to re-engineer such Java code may arise from the need to observe an agent-based system in operation or to integrate an agent component that has been obtained from a possibly unreliable source (e.g., the Internet). Java object code can be instrumented easily using the Jtrek tool (Compaq, 2001) which does not require the source code to be available.

- The approach is likely to succeed when re-engineering object-oriented code because it is then straightforward to set up the classes appropriately in the sequence diagrams (execution traces). This class structure will not be available in non-object oriented code, however, so a substantial amount of manual work would have to be done in order to obtain reasonable traces.
- The traces \mathcal{T} are obtained by running the instrumented code on various sequences of input. It is only these test inputs that are represented in the behavioral model generated. On the positive side, this gives the user an easy way of extracting from the code only the information in which (s)he is interested. On the negative side, the model generated will be incomplete. By using traditional techniques to generate test cases with a certain coverage, however, the model can be shown also to possess a certain coverage.
- A realistic system might produce a huge amount of different traces. These, however, might represent small variants of a few traces. Therefore, traces need to be abstracted before they can be used for synthesizing a new statechart. As an example, in the loading dock domain, an agent may rotate by n degrees. For the purposes of modeling, however, only the principal directions e , w , n and s are relevant and so the execution traces must be abstracted in order to avoid overly detailed models.
- Given the set of traces \mathcal{T} , for maximum effectiveness, OCL constraints should be set up. This is most likely a manual activity, although it may be possible to partially automate the selection of state variables by instrumenting the code appropriately.

Our work on the reverse engineering aspect is at an early stage. Initial results, however, have shown that there is a great deal of potential in this technique.

6. Conclusions

This chapter has presented an algorithm for automatically synthesizing UML statecharts from a set of sequence diagrams. For the development of large-scale agent-based systems, sequence diagrams can be a valuable means to describe inter-agent communication. Sequence diagrams can be extended with additional language constructs to enable generalizations and can be augmented with communication pre- and post-conditions in OCL. This enables the automatic detection and reporting of conflicts and inconsistencies between different sequence diagrams with respect to the pre/post-condition constraints. These annotations are furthermore used in the synthesis algorithm to correctly identify similar states and to merge a number of sequence diagrams into a single statechart. In order to make the algorithm practical, techniques for introducing hierarchy automatically into the generated statechart are employed.

A prototype of this algorithm has been implemented in Java and so far used for several smaller case-studies in the area of agent-based systems, classical object-oriented design (Whittle and Schumann, 2000), and human-computer interaction. In order to be practical for applications on a larger scale, the algorithm is being integrated into state-of-the-art UML-based design tools by way of an XMI interface.

This chapter has also discussed a novel application of the synthesis algorithm in the reverse engineering of existing systems. By simulating an agent-based system on a number of test cases, and instrumenting the code to output appropriate execution traces, these traces can be used as input to the synthesis algorithm and a behavioral model of the software can be extracted. This technique could have applications in understanding existing systems for which documentation no longer exists or which have been obtained by a possibly unreliable means (e.g., the Internet).

The synthesis algorithm presented in this chapter only describes the forward or synthesis part of the design cycle: given a set of sequence diagrams, we generate a set of statecharts. For full support of our methodology, research and development in two directions are of major importance: conflicts detected by the algorithm must not only be reported in an appropriate way to the designer but also should provide explanation on what went wrong and what could be done to avoid this conflict. We will use techniques of model-generation, abduction, and deduction-based explanation generation to provide this kind of feedback.

The other major strand for providing feedback is required when the user, after synthesizing the statechart, refines it or makes changes to

the statechart. In that case, it must be checked if the current statechart still reflects the requirements (i.e., the sequence diagrams), and in case it does, must update the sequence diagrams (e.g., by adding new communication acts).

The question whether UML (or AUML) is an appropriate methodology for the design of large-scale agent-based systems must still be answered. A part of the answer lies in the availability of powerful tools which support the development of agents during all phases of the iterative life-cycle. We are confident that our approach to close the gap between requirements modeling using sequence diagrams and design with statecharts will increase acceptability of UML methods and tools for the design of agent-based systems.

Acknowledgments

This work is supported by the NASA, grant 749-10-11 (Thinking Systems / Program Synthesis). We also want to thank the anonymous referees for their helpful suggestions.

Notes

1. `tm` is a timeout
2. In statecharts, a transition is labeled by e/a which means that this transition can be active only if event e occurs. Then, the state changes and action a is carried out. We use a similar notion in our definition of FSMs.
3. For visualization of the synthesized statechart we are using the tools Dot (Koutsofios and North, 1996) and daVinci (Fröhlich and Werner, 1994).

References

- Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley.
- AUML (2001). Agent UML. <http://www.auml.org>.
- Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. (1999). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.
- Bhatt, S. N. and Leighton, F. T. (1984). A Framework for Solving VLSI Graph Layout Problems. *J. Comp. Syst. Sci.*, 28:300–343.
- Biermann, A. and Krishnaswamy, R. (1976). Constructing Programs from Example Computations. *IEEE Transactions on Software Engineering*, SE-2(3):141–153.
- Bradshaw, J. (1997). *Software Agents*. American Association for Artificial Intelligence / MIT Press.
- Breu, R., Grosu, R., Hofmann, C., Huber, F., Krüger, I., Rumpe, B., Schmidt, M., and Schwerin, W. (1998). Exemplary and Complete Object Interaction Descriptions. In *Computer Standards and Interfaces*, volume 19, pages 335–345.
- Caire, G., Leal, F., Chainho, P., Evans, R., Garijo, F., Gomez, J., Pavon, J., Kearney, P., Stark, J., and Massonet, P. (2001). Agent oriented analysis using MESSAGE/UML. In Ciancarini, P. and Wooldridge, M., editors, *Agent Oriented Software Engineering*, pages 101–108, Berlin. Springer-Verlag.
- Castellø, R., Mili, R., and Tollis, I. G. (2000a). An algorithmic framework for visualizing statecharts. In Marks, J., editor, *Graph Drawing*, volume 1984 of LNCS, pages 139–149. Springer-Verlag.
- Castellø, R., Mili, R., Tollis, I. G., and Benson, V. (2000b). On the automatic visualization of statecharts: The vista tool. In *Formal Methods Tools 2000*.
- Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. (1994). *Object-Oriented Development: the FUSION Method*. Prentice Hall International.
- Compaq (2001). Jtrek. <http://www.compaq.com/java/download/jtrek>.

- Fröhlich, M. and Werner, M. (1994). Demonstration of the interactive graph-visualization system *davinci*. In Tamassia, R. and Tollis, I. G., editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 266–269. DIMACS, Springer-Verlag.
- Gehrke, T. and Firley, T. (1999). Generative sequence diagrams with textual annotations. In Spies and Schätz, editors, *Formale Beschreibungstechniken für verteilte Systeme (FBT99) (Formal Description Techniques for Distributed Systems)*, pages 65–72, München.
- Glinz, M. (1995). An integrated formal model of scenarios based on statecharts. In *5th European Software Engineering Conference (ESEC)*, volume 989 of *Lecture Notes in Computer Science*, pages 254–271, Springer-Verlag.
- Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.
- Khriss, I., Elkoutbi, M., and Keller, R. (1999). Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams. In Bezivin, J. and Muller, P., editors, *UML98: Beyond the Notation*, volume 1618 of LNCS, pages 139–149. Springer-Verlag.
- Koutsofios, E. and North, S. (1996). Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA.
- Leue, S., Mehrmann, L., and Rezaei, M. (1998). Synthesizing software architecture descriptions from Message Sequence Chart specifications. In *Proc. Thirteenth International Conference on Automated Software Engineering*, pages 192–195, IEEE Press.
- Männistö, T., Systä, T., and Tuomi, J. (1994). SCED report and user manual. Report A-1994-5, Dept of Computer Science, University of Tampere. ATM example available with the SCED tool from <http://www.cs.tut.fi/~tsysta/sced/>.
- Müller, J. (1996). *The Design of Intelligent Agents*. volume 1177 of LNAI. Springer-Verlag.
- Odell, J., Van Dyke Parunak, H., and Bauer, B. (2001). Representing Agent Interaction Protocols in UML. In Ciancarini, P. and Wooldridge, M., editors, *Agent Oriented Software Engineering*, volume 1957 of LNCS, pages 121–140. Springer-Verlag.
- OMG (2001). Unified Modeling Language specification version 1.4. Available from The Object Management Group (<http://www.omg.org>).
- Rao, A. S. and Georgeff, M. P. (1995). BDI Agents: From Theory to Practice. In Lesser, V., editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319. The MIT Press.
- Rosenberg, D. and Scott, K. (1999). *Use Case Driven Object Modeling with UML*. Object Technology Series. Addison Wesley.

- SBSE (2000). Workshop on scenario-based round trip engineering.
<http://www.cs.uta.fi/~cstasy/oops1a2000/workshop.html>.
- Schumann, J. (2000). Automatic debugging support for uml designs. In Ducasse, M., editor, *Proceedings of the Fourth International Workshop on Automated Debugging*. <http://xxx.lanl.gov/abs/cs.SE/0011017>.
- Singh, M. (1998a). A customizable coordination service for autonomous agents. In *Intelligent Agents IV: 4th International Workshop on Agent Theories, Architectures, and Languages*.
- Singh, M. (1998b). Developing formal specifications to coordinate heterogeneous autonomous agents. In *International Conference on Multi Agent Systems*, pages 261–268.
- Somé, S. and Dssouli, R. (1995). From scenarios to timed automata: building specifications from users requirements. In *Asia Pacific Software Engineering Conference*, pages 48–57.
- Systä, T. (2000). Incremental construction of dynamic models for object oriented software systems. *Journal of Object Oriented Programming*, 13(5):18–27.
- van Lamsweerde, A. (1998). Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114.
- Warmer, J. and Kleppe, A. (1999). *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Object Technology Series. Addison-Wesley.
- Whittle, J. and Schumann, J. (2000). Generating Statechart Designs From Scenarios. In *Proceedings of International Conference on Software Engineering (ICSE 2000)*, pages 314–323, Limerick, Ireland.
- Wooldridge, M. and Ciancarini, P. (2001). Agent-oriented software engineering: The state of the art. In *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Co.
- Wooldridge, M. and Jennings, N. (1995). Intelligent agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152.
- Wooldridge, M., Jennings, N., and Kinny, D. (1999). A methodology for agent-oriented analysis and design. In *Third International Conference on Autonomous Agents (Agents 99)*, pages 69–76, Seattle, WA.